Determining Coefficients of Checking Polynomials for an Algebraic Method of Fault Tolerant Computations of Numerical Functions

A Thesis
Presented to
The Academic Faculty

by

Clinton C Jones

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
April 2004

Determining Coefficients of Checking Polynomials for an Algebraic Method of Fault Tolerant Computations of Numerical Functions

Approved by:

Dr. Douglas Blough, Professor ECE

Dr. Benjamin Klein, Assistant Professor ECE

Dr. Tuna Tugcu, Visiting Assistant Professor ECE

Dr. Feodor Vainstein, Thesis Advisor, Professor ECE

Date Approved: April 5, 2004

ACKNOWLEDGEMENT

My sincere thanks to Dr. Feodor Vainstein for introducing me to the field of fault tolerant computing, for serving as my academic advisor, and for research support at Georgia Tech; to Dr. Douglas Blough, Dr. Benjamin Klein, and Dr. Tuna Tugcu for being a part of my reading committee; to Dr. David Frost and the staff at Georgia Tech Savannah for allowing a flexible work schedule; to my parents for a lifetime of guidance and encouragement; and, to the greatest engineer and blessing I've ever known, my wife Amy – whose love, support, and friendship have made this and so much more in my life possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABFT	Algorithm based fault tolerance
ASIC	Application specific integrated circuit
COTS	Commercial off-the-shelf
CPLD	Complex programmable logic device
DSP	Digital signal processing
FPGA	Field-programmable gate array
GUI	Graphical user interface
LC	Linearly checkable
LSE	Least square estimation
LSEFUNRUN	Matlab function for finding polynomial coefficients
PC	Polynomially checkable
PPC	Partially polynomially checkable
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit

SUMMARY

This thesis presents a means for determining checking polynomials for the fault tolerant computation of numerical functions. This method is based on certain algebraic features of the numerical functions such as the transcendence degree of a field extension. Checking polynomials are given for representative simple and compound numerical functions. Some of these checking models are then implemented in a simulation environment. The program developed provides the means for generating checking polynomials for a broad class of numerical functions. Considerations for designing and deploying checking models are given. This numerical technique can lower costs and conserve system resources when engineering for remote or nanoscale supercomputing environments.

Chapter 1

Introduction

1.1 Fault tolerant computing

The fault tolerance of a system is a measure of how gracefully it responds to the unexpected. Most modern high-performance and critical systems in industrial, scientific, and consumer markets are controlled by computers. While these areas are becoming more dependent on computing, the computer itself is evolving from a system with a central processing unit on a single microchip towards more complex and integrated system-on-a-chip (SOC) architectures. The converging demands on computing, coupled with increasing expectations, dictate that mission or safety-critical systems exhibit a high level of fault tolerance [Weinstock 97]. By responding gracefully to unexpected events, the fault tolerant system may preserve property, critical information, or life itself.

Rapidly improving manufacturing and microprocessor performance has led system designers to expect ever smaller computing devices to manage numerical computations on a scale once reserved only for supercomputers [Turmon 00]. Detecting, locating, and correcting errors that may occur within those numerical computations then becomes a critical aspect of the overall microprocessor or SOC design. While computer system designers recognize that fault tolerance is important they must also recognize that any fault tolerant scheme should be sensitive to time and space constraints. Fault tolerant computing, as an ideal, should achieve the detection and correction of errors from many potential sources: software faults, hardware imperfections, random failures, and environmental interference.

1.2 Contribution of thesis

This thesis presents an algorithm for determining checking polynomials for computing numerical functions. This method is based on the generalization of the addition theorem presented by F. Vainstein [Vainstein 91]. It requires smaller redundancy, provides better fault coverage, better opportunity for fault location, and is substantially more general than other known methods of achieving fault tolerance for numerical computations.

A program is presented here which will allow a system designer to determine the checking polynomial for a broad class of numerical functions. Through the use of this program a short library of checking polynomials for some common numerical functions is given. In addition, examples are shown of how a system designer could deploy numerical functions with this method.

1.3 Organization of thesis

In Chapter 2, the backdrop of fault tolerant computing is given and the major alternative approaches are noted. In Chapter 3, the main theory behind this paper is presented. This includes some background from Algebra and F. Vainstein's main result. Chapter 4 presents the significant contribution of this thesis – the program that generates the coefficients of a checking polynomial for many numerical functions. This includes some discussion of the Matlab® environment and why it was chosen for this project. In Chapter 5, a short library of checking polynomials for some common numerical functions is presented. In addition, some more "complicated-looking" functions are given as an example of further results. In Chapter 6, an example is given of how a checking polynomial can be integrated with the implementation of a numerical function in a

system design. Finally, in Chapter 7, some conclusions are discussed along with a mention of possible future directions for this research.

Chapter 2

Historical Perspective

2.1 Reliability from unreliable components

The prolific mathematician Von Neumann was among the first to consider the growing problem of how to build a reliable computing system from unreliable components [von Neumann 56]. We can say that a fault tolerant system is generally realized in four stages: detecting errors, confining damage, recovering from the error, and treating the fault [NIST 95]. The measure to which a system meets these objectives, or recovers from faults, is measured by its fault coverage.

There have been two major avenues of approach to the problem of detecting and correcting errors in computing systems. The first approach views the result of the computation or the data stored in memory as independent and introduces modular space or time redundancy into the design. The second approach makes use of specific properties of the computational function to exploit hidden or coded redundancy.

2.2 Different approaches

2.2.1 Space and time redundancy

An obvious solution to fault tolerance and the simplest in terms of implementation is the use of space or time redundancy. One example, shown below in Figure 2.1, of the space redundancy approach, is replication with voting [Parag 85] where identical modules or

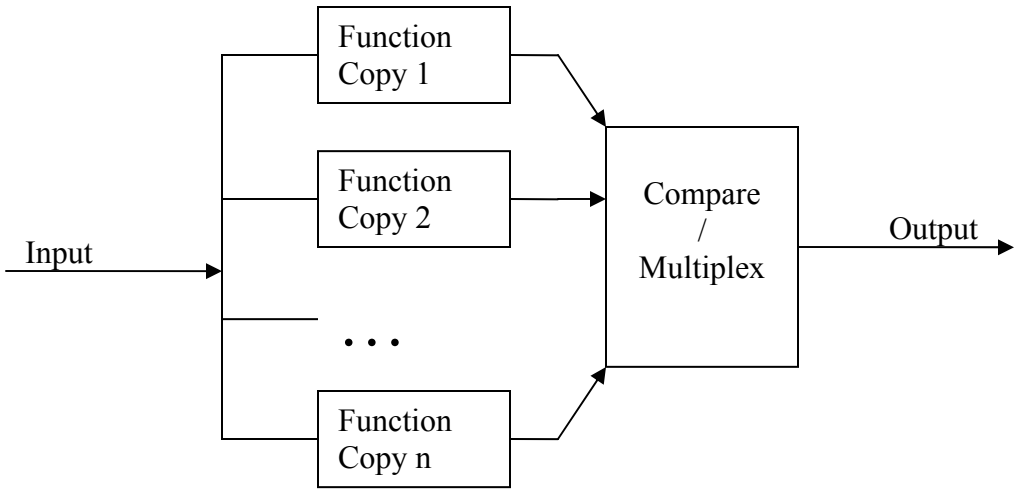


Figure 2.1 The fault tolerant strategy of function replication with voting.

processors concurrently execute the function and vote to determine the likely result. Another example is time redundancy where the function is repeatedly executed by the same processor, storing the result in each case, and comparing the outputs after some set number of iterations. These methods, although widely used, have a large impact on the size of the design or processor execution time [Esonu 94, Hosseini 89, Kuo 92, Majumdar 90, Singh 88].

2.2.2 Codes and algorithms

Error-correcting codes [Asumth 82, Blakely 85, Siewiorek 82], probabilistic methods [Blum 88,90], interval arithmetic methods [Alefeld 83, Moore 79 Neumaier 90], and algorithm-based fault tolerance techniques [Huang 84, Vijay 97] employ redundant information in some form to detect errors. Either redundant information is pre-coded in the information prior to processing and then decoded or, as in the case of algorithm-based fault tolerance, redundant computations within the algorithms are exploited. Linear checks for polynomials described in [Karpovsky 79-82] are limited to polynomial

functions only. This approach can be used only for functions for which functional equations are known. This limitation was overcome by the work of F. Vainstein who showed that functional equations exist for a very broad set of functions [Vainstein 91].

2.2.3 Checking polynomials

The innovative fault tolerant strategy presented in [Vainstein 93] is based on a generalization of the classical addition theorem from algebra. This theorem is recognizable, for example, in the addition formula $e^{x+y} = e^x e^y$. More generally, the function f satisfies an algebraic addition theorem if $A(f(x), f(y), f(x+y))$ is identically zero for some polynomial $A(x, y, z)$.

The outcome of the fault tolerant strategy based on this simple idea is different in some important ways from those mentioned above. It employs the specific structure of the function to be computed and is based on certain algebraic concepts such as transcendental degree of field extension. Important advantages are realized by this method including smaller redundancy, better fault coverage, better opportunity for fault location, and more generality than methods such as the similarly named linear checking polynomial method mentioned above.

Chapter 3

Polynomial Checking

3.1 Example of polynomial checking

I will follow the work of [Vainstein 93] to introduce checking polynomials. Before diving into the background theory it is worth considering an example of the method of testing numerical functions with checking polynomials.

Example: Suppose we wish to compute and check the function

$$f(x) = e^{-x}\sin 5x, x \in [0,10]$$

Let $a_1, a_2 \in R$; and denote by

$$f_0 = f(x+0) = e^{-x}\sin 5x,$$

$$f_1 = f(x+a_1) = e^{-(x+a_1)}\sin 5(x+a_1),$$

$$f_2 = f(x+a_2) = e^{-(x+a_2)}\sin 5(x+a_2).$$

Denote by $p_1 = e^{-a_1}\cos 5a_1$; $q_1 = e^{-a_1}\sin 5a_1$; $p_2 = e^{-a_2}\cos 5a_2$; $q_2 = e^{-a_2}\sin 5a_2$;

$$A = p_1q_2 - p_2q_1;\ B = -q_2;\ C = q_1$$

Then $Af_0 + Bf_1 + Cf_2 = 0$ for every $x \in R$. (1)

It is very important that A, B and C do not depend on x and depend only on a_1 and a_2 Taking (1) into consideration we can consider the following method for error detection.

Denote the computed values of function f at the points $x, x+a_1, x+a_2$ by $\tilde{f}_0, \tilde{f}_1, \tilde{f}_2$ respectively. Then if the computation is correct

$A\tilde{f}_0 + B\tilde{f}_1 + C\tilde{f}_2 = 0$ (Independently of x) (2)

Now consider that we want to correct a single error.

Consider $a_1 = a;\ a_2 = 2a$ and let $A\tilde{f}_0 + B\tilde{f}_1 + C\tilde{f}_2 \neq 0$ (3)

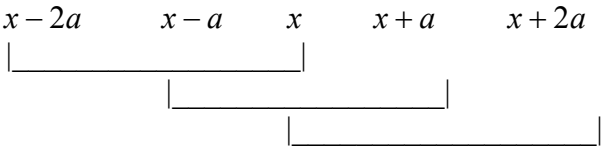
Because one of $\tilde{f}_i \neq f_i;\ i = 0, 1, 2.$

Suppose for example that $\tilde{f}_0 \neq f_0;\quad \tilde{f}_1 = f_1;\ \ \tilde{f}_2 = f_2$

Then the correct value is given by the formula

$$f_0 = -\frac{B}{A}\tilde{f}_1 - \frac{C}{A}\tilde{f}_2 \tag{4}$$

Location of the error can be obtained by using (2) for the following triples:



It should be taken into consideration that computations are done in practice with a certain level of accuracy. Hence the formula (2) should be substituted by the formula

$$|\ A\tilde{f}_0 + B\tilde{f}_1 + C\tilde{f}_2\ | \leq \delta, \tag{2'}$$

where δ is a small positive number specified by the precision of the computation.

3.2 Field extension theory

Field extension theory is a subset of algebra. Background information about fields and field extensions can be found in Lang [Lang 65] or online at [Wiesstein 99] and is provided here for the reader's convenience. Proofs of stated theorems and corollaries are given when these add to an understanding of the theory. For a full development see [Vainstein 93].

Definition: K is called a **subfield** of L, if a subset K of the elements of field L satisfies the field axioms with the same operations of L. A field L is a **field extension** (or **extension field**) denoted $K \subset L$, of a field K if K is a subfield of L. For example, the complex numbers are an extension field of the real numbers, $R \subset C$.

Definition: Let $K \subset L$ be a field extension and $K[T_1,....,T_n]$ be the set of all polynomials in $T_1,...,T_n$ over K. The elements $a_1,...,a_n \in L$ are called **algebraically dependent** over K, if there exists a polynomial $P \in K[T_1,...,T_n], P \neq 0$, such that $P(a_1,..,a_n) = 0.$ The elements $a_1,...,a_n \in L$ are called **algebraically independent** over K, if they are not algebraically dependent. By $K(T_1,...,T_n)$ we denote the quotient field of the ring $K[T_1,...,T_n]$.

Example: Consider the field extension $Q \subset R.$ The numbers $\sqrt{2}$ and $\sqrt{3} \in R$ are algebraically dependent over Q. $P(T_1,T_2) = T_1^2 + T_2^2 - 5.$ While the numbers 1 and $\pi \in R$ are algebraically independent over Q.

Definition: Let $K \subset L$ be a field extension. The **transcendental degree** of this extension is by definition the maximum possible number of elements from L algebraically independent over K.

If the transcendental degree of $K \subset L$ is equal to n and $m>n,$ then any subset $\{ a_1,...,a_m \} \subset L$ is algebraically dependent.

Example: The transcendental degree of $R \subset R(T)$ is 1.

The transcendental degree of $R \subset R(x,e^x)$ is 2.

The transcendental degree of $R(x,\sin x,e^x) \subset R(x,\sin x,\cos x,e^x)$ is 0.

3.3 Checking polynomials

A function $f:R \to R$ is called **polynomially checkable** (PC) if there exists an integer *k*, such that for any $a_1,...,a_k \in R$ the functions

$$f_0(x) = f(x), f_1(x) = f(x + a_1),..., f_k(x) = f(x + a_k)$$

are algebraically dependent, i.e. there exists a polynomial $P \in R[T_0,...,T_k]$, such that $P(f_0,..., f_k) = 0$ (for any $x \in R$). The polynomial *P* is called a **checking polynomial** of the function *f*.

The computation of a PC function can be readily verified. For a given value of *x*, denote by $\tilde{f}_0, \tilde{f}_1,...., \tilde{f}_k$ the values of *f* at the points $x, x + a_1,..., x + a_k$, respectively. Then if all the values are computed correctly, the following equality holds:

$$P(\tilde{f}_0, \tilde{f}_1,..., \tilde{f}_k) = 0 \qquad (5)$$

This property provides a unified approach to the problem of error detection/correction in computation of numerical functions. We can consider the inequality

$$\left|P(\tilde{f}_0, \tilde{f}_1,..., \tilde{f}_k)\right| \le \delta, \qquad (5')$$

where δ is a small positive number specified by the precision of the computation. In the case of a correct computation (5') is satisfied. Note, however, that even if (5') is satisfied it doesn't give a 100% guarantee of correct computation. There are some faults that cannot be detected by (5').

Example: Consider an example where we denote by *S* the set of functions: $x, e^x, \sin x$.

Let $A \subseteq S$; denote by R(A) the field of all rational functions in $\boldsymbol{g}_j \in \boldsymbol{A}$ and by $\overline{R(A)}$ its algebraic closure.

a) A = (x), R(A) = $\left\{\dfrac{P_i(x)}{Q_i(x)}\right\}$, where P_i, Q_j are polynomials of one variable with real coefficients. The algebraic closure $\overline{R(A)}$ includes, as a special case, any function *g(x)* which is a solution of an equation $P_n(x)g^n(x) + P_{n-1}(x)g^{n-1}(x) + ... + P_0(x) = 0$ where $P_i(x), i = 0,1,...,n$ are polynomials of one variable with real coefficients. In particular, $\overline{R(A)}$ includes the set of all functions that can be obtained by application of a finite number of additions, subtractions, multiplications, divisions, and rational powers of the function $g(x) = x$.

b) $A = \{e^x, \sin x\}$; $R(A) = \dfrac{P_i(e^x, \sin x)}{Q_i(e^x, \sin x)}$, where P_i, Q_j are polynomials of two variables with real coefficients.

Theorem: Let $f: R \to R$ belong to the field $\overline{R(A)}, A \subseteq (x, e^x, \sin x)$. Then *f* is **polynomially checkable** with $k = |A|$.

Proof: [Vainstein 91] proved for the case $A = \{x, e^x, \sin x\}$. For the other cases the proof is analogous.

Let $f(x) \in \overline{R(x, e^x, \sin x)}$ and $a_1, a_2, a_3 \in R$; denote:

$$f_0(x) = f(x),\ f_1(x) = f(x + a_1),\ f_2(x) = f(x + a_2),\ f_3(x) = f(x + a_3).$$

We have to show that $f_0,...,f_3$ are algebraically dependent. This follows from the statements:

1) The transcendental degree of $R \subset \overline{R(x,e^x,\sin x)}$ equals to 3.

2) For every $a \in R, f(x+a) \in \overline{R(x,e^x,\sin x)}$.

Indeed $f(x) \in \overline{R(x,e^x,\sin x)} \Leftrightarrow$ there exists a polynomial $A \in R(x,e^x,\sin x)[T]$, such that $A(f) = A_n(x)f(x) + ... + A_1(x)f(x) + A_0(x) = 0$, where $A_i(x) \in R(x,e^x,\sin x)$.

If we denote $\rho(x) = A_n(x)f(x) + ... + A_0(x)$, then

$\rho(x+a) = A_n(x+a)f(x+a) + ... + A_0(x+a) = 0, A_i(x+a) \in R(x,e^x \sin x, \cos x)$.

Hence, $f(x+a) \in \overline{R(x,e^x,\sin x,\cos,)}$. But $\overline{R(x,e^x,\sin x,\cos x)} = \overline{R(x,e^x,\sin x)}$, hence $f(x+a) \in \overline{R(x,e^x,\sin x)}$ and, therefore, $f_0,f_1,f_2,f_3 \in \overline{R(x,e^x,\sin x)}$. But the transcendental degree of $R \subset \overline{R(x,e^x,\sin x)}$ equals to 3, therefore $f_0 f_1,f_2,f_3$ are algebraically dependent.

Let f be the result of the application of a finite number of additions, subtractions, multiplications, divisions, and rational powers of the following functions:

$$Const,\ x, e^x, \sin(r_i x + b_i), \cos(r_j x + b_j),$$

where r_i, r_j are rational numbers.

Then f is a PC function where $k \le 3$.

Example: The function $f(x) = \dfrac{(\sin(\frac{x}{11} + \frac{\pi}{7}) + e^x)^{\frac{3}{5}} + x^2 \cos^4 x}{x^5 + (x^4 + x^2(\sin 2x + xe^x)^3)^{\frac{1}{3}}}$ is a PC function with k=3.

Example: Consider the function

$$f(x) = \frac{((\sin x)^{\frac{1}{3}} + \cos x)^{\frac{1}{2}}}{\sin(x+7) - (\cos 3x \sin(3x+5) - 4)^{\frac{1}{17}}}; f(x) \in \overline{R(\sin x)}$$

The transcendental degree of extension $R \subset \overline{R(\sin x)}$ is 1, therefore *f(x)* is a PC function with *k*=1.

The theorem above indicates that the class of PC functions is quite large. Note, however, that some commonly used functions including $\log(x), \sin^{-1}(x)$, and $\cos^{-1}(x)$ are non PC functions.

3.4 Fault classification

Definition: The first class of faults we may consider can be called **software faults**. These occur if some other PC function $g(x) \neq f(x)$ has the same checking polynomial. For example if $g(x) = f(x+b),$ where b is a constant, then g(x) and f(x) have the same checking polynomials. [Vainstein 91] shows that a PC function with a bounded spectrum is uniquely defined by its checking polynomial (the set of shifts is fixed) and its values are at a finite set of points. This property can be used to confront the software faults.

Definition: The second class of faults detectable by using this method can be called **hardware faults.** These are a result of physical defects in the device which performs the calculation of the function. Random faults are classified as hardware faults. It is shown by [Vainstein 91] that the class of PC functions is very broad even for a small *k* in (5).

3.5 Linear checks

Definition: A function $f : R \to R$ is called **linearly checkable (LC)** if there exists an integer *k* such that for any $a_1, \ldots, a_k \in R$ the functions

$$f_0 = f(x), f_1 = f(x + a_1), \ldots, f_k = f(x + a_k)$$

are linearly dependent.

Any LC function is a PC function for which there exists a checking polynomial of degree equal to 1.

If $f : R \to R$ satisfies a linear differential equation with constant coefficients

$$c_m \frac{d^m f}{dx^m} + c_{m-1} \frac{d^{m-1} f}{dx^{m-1}} + \cdots + c_0 = 0 \qquad (6)$$

then *f* is an LC function with $k = m$.

If $f : R \to R$ is a differentiable LC function with a given *k* then *f* is infinitely differentiable and satisfies a linear differential equation with constant coefficients (6), where $m \le k$.

Consider $S(f) = \{f(x + a) \mid a \in R\}$ the set of all functions obtained from the function *f* by shifting of the argument. Denote by dim*S*(*f*) the maximum possible cardinality of a subset of linearly independent functions from *S*(*f*). Denote dim*S(f)* by *m* and let $g_1, \ldots, g_m \in S(f)$ be linearly independent. Denote by $LS(f) = Rg_1 + \cdots + Rg_m$. Then *LS*(*f*) is a linear space and $S(f) \subset LS(f)$.

Definition: Let *A* be any algebra over R, and $B \subset A$. Denote by $\text{tr.deg}_R(B)$ the maximum possible cardinality of a subset of algebraically independent elements from *B*. If $f : R \to R$ then denote by $\text{tr.deg}_R(f) = \text{tr.deg}_R(S(f))$. Denote by $\dim(f) = \dim(LS(f))$

3.6 Degree of a checking polynomial

The definition of a PC function can be extended to include complex valued functions. This not only generalizes the class of functions but also simplifies the evaluation of the degree of a checking polynomial.

Definition: A function $f : R \to R$ is called **polynomially checkable (PC)** if there exist a number *k* such that for any $a_1, \ldots, a_k \in R$, the functions

$f_0 = f(x), f_1 = f(x + a_1), \ldots, f(x + a_k)$ are algebraically dependent over **C**.

If *Y* is a field of functions $\varphi : R \to C$ such that for any $a \in R$ and any $\varphi \in Y$ the function $\varphi(x + a) \in Y$. Let *k*= tr.deg$_C(Y) < \infty$ and $f \in \tilde{Y}$ then *f* is a PC function with this *k*. If for *f* there exist such transcendental basis $y_1, \ldots, y_k$ of *Y* and such $a_1, \ldots, a_k \in R$ that the following polynomial equations are satisfied

$$P_0(y_1, \ldots, y_k, f_0) = f_0^{d_0} + f_0^{d_0 - 1} A^0_{d_0 - 1}(y_1, \ldots, y_k) + \ldots + A^0_0(y_1, \ldots, y_k) = 0$$
$$P_1(y_1, \ldots, y_k, f_1) = f_1^{d_1} + f_1^{d_1 - 1} A^0_{d_1 - 1}(y_1, \ldots, y_k) + \ldots + A^1_0(y_1, \ldots, y_k) = 0$$
$$\vdots$$
$$P_k(y_1, \ldots, y_k, f_k) = f_k^{d_k} + f_k^{d_k - 1} A^0_{d_k - 1}(y_1, \ldots, y_k) + \ldots + A^k_0(y_1, \ldots, y_k) = 0$$

where $A^i_j \in C[y_1, \ldots, y_k]$. Then, for *f* there exists a checking polynomial *P* such that deg $P \le \prod_{i=0}^{k} d_i$. In other words, there exists an upper bound on the degree of a checking polynomial. This and other theorems stated above are proved in [Vainstein 93].

3.7 Other cases

Other cases for checking polynomials are described in [Vainstein 93]. These include partially polynomially checkable functions, PC functions of several variables, LC

functions of several variables, and error detecting faults of different multiplicity. These cases are worth further consideration in their own right but are not covered here.

Chapter 4

Finding Checking Polynomials

4.1 How to find a checking polynomial

Given a function $f : R \to R$ the first step is to consider the set $S(f) = \{f(x+a) \mid a \in R\}$. Denote by $B(f)$ the smallest field of real functions having $S(f)$ as a subset and by *k* the transcendental degree of $R \subset B(f)$. If *k* is finite, then *f* is a PC function, and there exists a checking polynomial for it with the number of variables equal to $k+1$.

Our task is to find the coefficients of $P(T_0, \ldots, T_k)$. If we know deg*P* then the coefficients can be determined by the method of indefinite coefficients. The equality $P(f(x_i), f(x_i + a_1), \ldots, f(x_i + a_k)) = 0$ for every $x_i \in R$. A system of linear equations can be formed then by choosing different $x_i's$. Then, the coefficients of the checking polynomial are found as the solution of that system.

4.1.1 Example

Consider $f(x) = \frac{e^x}{x}$. $S(f) = \left\{ \frac{e^{x+a}}{x+a} \mid a \in R \right\}$. The transcendental degree of $R \subset B(f)$ equals to 2, therefore $k = 2$. The degree of the checking polynomial *d* is an unknown.

Starting with $d = 1$ and $a_1 = 1, a_2 = 2$. Then $f_0(x) = \frac{e^x}{x}$, $f_1(x) = \frac{ee^x}{x+1}$, $f_2(x) = \frac{e^2 e^x}{x+2}$.

For $x_i = i$ we can consider linear equations of the form $Af_0(x_i) + Bf_1(x_i) + Cf_2(x_i) = 0$ with unknowns *A*, *B*, and *C*. This leads to consideration of the system of infinitely many equations:

$$Ae + B\frac{e^2}{2} + C\frac{e^3}{3} = 0$$
$$A\frac{e^2}{2} + B\frac{e^3}{3} + C\frac{e^4}{4} = 0$$
$$\vdots$$
$$A\frac{e^i}{i} + B\frac{e^{i+1}}{i+1} + C\frac{e^{i+2}}{i+2} = 0$$
$$\vdots$$

The infinite system is inconsistent. However, considering $d = 2$, after computations we find the checking polynomial

$P(T_0, T_1, T_2) = 2eT_0T_2 - T_1T_2 - e^2T_0T_1$.

4.2 Application of least square estimation

This process may be realized in the following optimization problem. Let

$$f : [A, B] \to R$$

Denote

$$\delta(\beta_0, \alpha_1, \ldots, \alpha_k) = \int_A^B (f(x) - \alpha_1 f(x + a_1) - \ldots - \alpha_k f(x + a_k) - \beta_0)^2 dx$$

And find $\alpha_1, \ldots, \alpha_k, \beta_0$, so that $\delta(\beta_0, \alpha_1, \ldots, \alpha_k)$ takes minimal value.

To solve this problem consider the equations:

$$\frac{\partial}{\partial \beta_0}\delta = \int_A^B 2(f(x) - \alpha_1 f_1 - \ldots - \alpha_k f_k - \beta_0)(-1)dx = 0$$
$$\frac{\partial}{\partial \alpha_1}\delta = \int_A^B 2(f(x) - \alpha_1 f_1 - \ldots - \alpha_k f_k - \beta_0) f_1 dx = 0$$
$$\vdots$$
$$\frac{\partial}{\partial \alpha_k}\delta = \int_A^B 2(f(x) - \alpha_1 f_1 - \ldots - \alpha_k f_k - \beta_0) f_k dx = 0$$

Denote by $\langle f, g \rangle = \int_A^B (f \cdot g) dx$. Using this notation, we can express the system of equations in the form

$$\begin{aligned}
&\langle f_0, 1 \rangle = \alpha_1 \langle 1, f_1 \rangle + \ldots + \alpha_k \langle 1, f_k \rangle + \beta_0 (B - A) \\
&\langle f_0, f_1 \rangle = \alpha_1 \langle f_1, f_1 \rangle + \ldots + \alpha_k \langle f_1, f_k \rangle + \beta_0 \langle f_1, 1 \rangle \\
&\langle f_0, f_2 \rangle = \alpha_1 \langle f_2, f_1 \rangle + \ldots + \alpha_k \langle f_2, f_k \rangle + \beta_0 \langle f_2, 1 \rangle \\
&\vdots \\
&\langle f_0, f_k \rangle = \alpha_1 \langle f_k, f_1 \rangle + \ldots + \alpha_k \langle f_k, f_k \rangle + \beta_0 \langle f_k, 1 \rangle
\end{aligned}$$

Solving this system we obtain $\beta_0, \alpha_1, \ldots, \alpha_k$. If $\delta(\beta_0, \alpha_1, \ldots, \alpha_k) = 0$ then f is an LC function with the checking polynomial

$$f_0 - \alpha_1 f_1 - \ldots - \alpha_k f_k - \beta_0 = 0$$

If $\delta(\beta_0, \alpha_1, \ldots, \alpha_k) = \delta \neq 0$ then f does not have a checking polynomial of degree 1. However, if δ is a small number, the formula

$$\left| \tilde{f}_0 - \alpha_1 \tilde{f}_1 - \ldots - \alpha_k \tilde{f}_k - \beta \right| \leq \delta \qquad (4.2.1)$$

can be used to verify the correctness of computations. A similar method can be used for obtaining a checking polynomial of degree > 1.

Other methods for finding a checking polynomial are described in [Vainstein 93].

4.3 Algorithm implementation

A program was developed by the author to implement the optimization algorithm described above using techniques of linear algebra. This program determines the coefficients of the checking polynomial as defined by, $\beta_0, \alpha_1, \ldots, \alpha_k$, for any numerical function with checking inequality as defined in equation (4.2.1) above.

From the system of equations defined above, denote

$$A = \begin{pmatrix} \langle 1, f_1 \rangle & \cdots & \langle 1, f_k \rangle & (B - A) \\ \langle f_1, f_1 \rangle & \cdots & \langle f_1, f_k \rangle & \langle f_1, 1 \rangle \\ \langle f_2, f_1 \rangle & \cdots & \langle f_2, f_k \rangle & \langle f_2, 1 \rangle \\ \vdots & \vdots & \vdots & \vdots \\ \langle f_k, f_1 \rangle & \cdots & \langle f_k, f_k \rangle & \langle f_k, 1 \rangle \end{pmatrix}.$$

Define a vector X as

$$X = \begin{pmatrix} \beta_o \\ \alpha_1 \\ \vdots \\ \alpha_k \end{pmatrix}.$$

And, define a vector B as

$$B = \begin{pmatrix} \langle f_0, 1 \rangle \\ \langle f_0, f_1 \rangle \\ \langle f_0, f_2 \rangle \\ \vdots \\ \langle f_0, f_k \rangle \end{pmatrix}.$$

First, the program computes the coefficients of matrix A and vector B using a straight-forward trapezoidal integration method. Then, the equation $AX = B$ is solved by a reduced row echelon matrix form. The resulting values of vector X then give the coefficients of the checking polynomial, $\beta_0, \alpha_1, \ldots, \alpha_k$.

The values of $\beta_0, \alpha_1, \ldots, \alpha_k$ can then be used to evaluate the delta function $\delta(\beta_0, \alpha_1, \ldots, \alpha_k)$ as described above. This will give the deviation value of δ.

In order to investigate the effect of increasing *k* (as defined in the optimization algorithm), the program will also find the values of δ for a given range of k values and determine which of these produces the minimum deviation.

4.3.1 Example run of the program

The program is written as a Matlab function. Before discussing the reasons for implementing the algorithm in Matlab consider this example.

Example: Consider $f(x) = \sin(x)$. Suppose that we wish to determine a checking polynomial for f over $x \in [0,2\pi]$. The LSEFUNRUN (least square estimation function) program is invoked at the Matlab command line for this example by

[k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN('sin(x)',0.001,0,6.283,(1:5))

And returns the output

Table 4.1 The output of the program for the sine function.

k	3
delta	7.006×10^{-15}
alphas	0.681 -0.568 -0.400
betao	-2.664×10^{-16}
stepsize	0.001
A	0
B	6.283

The input parameters will be explained further below. However, the output in Table 4.1 identifies the values of the unknowns in the least square estimation algorithm above. The *k* indicates the degree of the best checking polynomial found in the interval $[A,B]$, *delta* is the deviation or error of the best checking polynomial, the *alphas* and *betao* are the coefficients of the checking polynomial, and the *stepsize* indicates the distance between

elements taken in the domain vector $[A, B]$. These output parameter definitions are summarized below in Table 4.2.

Table 4.2 Description of least square estimation function output.

k	The number of shifted functions in the checking polynomial.
delta	The error bound on the checking polynomial.
alphas	The checking polynomial coefficients.
betao	The constant term for the checking polynomial.
stepsize	The domain interval step size.
A	Lower bound on domain interval.
B	Upper bound on domain interval.

4.3.1.1 Understanding the plots

The program returns the plots shown below in Figure 4.2.

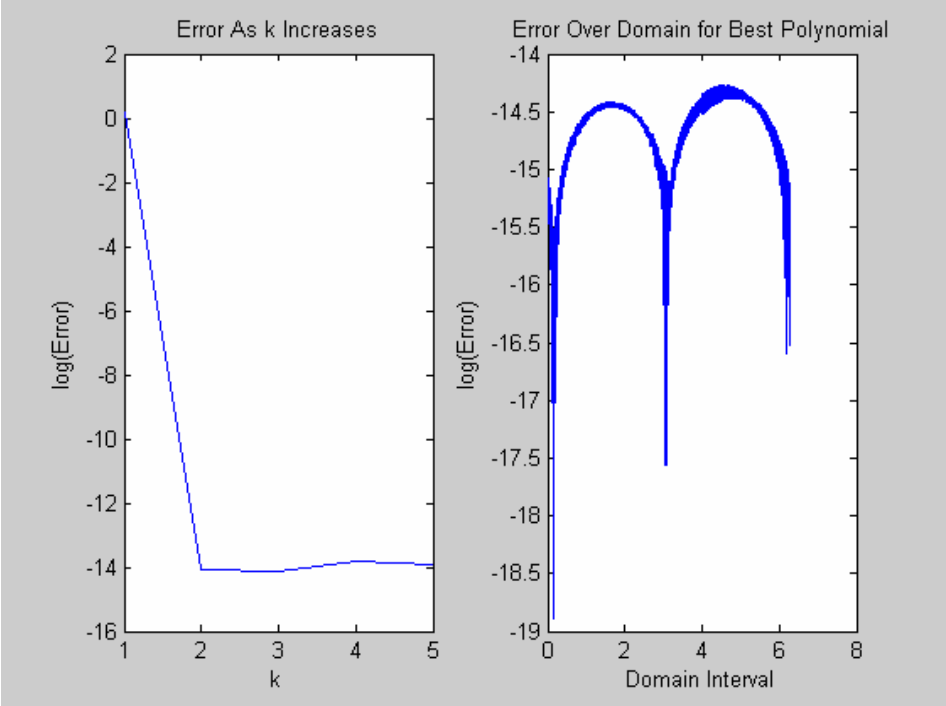


Figure 4.2 Plots of checking polynomial error for the sine function.

The plot on the left shows the log(accuracy), the delta function value, of the checking polynomial over increasing values of *k*. We can gather two important things from this

plot. First, this is an LC function because we see the delta function immediately falls to the Matlab arithmetic limit of about 10^{-30}. And, second, we note that checking polynomials for sin(x) for $k \geq 2$ have approximately the same accuracy. So, the gain in accuracy for $k > 2$ is negligible and would only result in unnecessary overhead. Therefore, the best choice for the checking polynomial would be $k = 2$.

The plot on the right gives the delta function values of the best *k* polynomial from the interval $k \in [1,\ldots,5]$ over the interval of the domain $x \in [0,2\pi]$. Careful inspection of the plot on the left will reveal that the delta function reaches an absolute minimum at $k = 3$. Thus, the program chooses this value to generate the plot on the right. The plot on the right then shows the upper bound on the error for sin(x) in $x \in [0,2\pi]$. We noted above that the left plot indicates that the best choice for *k* would be $k = 2$ to avoid unnecessary overhead. We also noted that the values for $k \geq 2$ remain very much the same – thus, we can have confidence that the error in the interval for $k = 2$ would be close to that shown for $k = 3$.

4.3.1.2 Understanding the output data

The program returns seven parameters: *k*, *delta*, *alphas*, *betao*, *stepsize*, *A*, *B*. Each of these relates directly to the parameters as defined in section 4.2 above – the application of least square estimation. Except for *stepsize*, each of these is shown in

$$\delta(\beta_0, \alpha_1, \ldots, \alpha_k) = \int_A^B 2\left(f(x) - \alpha_1 f(x + a_1) - \ldots - \alpha_k f(x + a_k) - \beta_0\right)^2 dx$$

where the vector $alphas = [\alpha_1,\ldots,\alpha_k]$ and *betao* form the coefficients of the desired checking polynomial. The values of these coefficients returned above is given in Table 4.3.

Table 4.3 Checking polynomial coefficients for the sine function.

alphas	0.681 -0.568 -0.400
betao	-2.664×10^{-16}

These coefficients allow us to construct the checking polynomial

f(x) = sin(x)-alphas(1)*sin(x+1)-alphas(2)*sin(x+2)-alphas(3)*sin(x+3)-betao

Indeed, if we check by hand the value of this checking polynomial for a random value of the domain, $x = 0.350$,

f(.35) = sin(.35)-alphas(1)*sin(.35+1)-alphas(2)*sin(.35+2)-alphas(3)*sin(.35+3)-betao

$$f(.35) = -7.744\times10^{-16}$$

Inspection of the plot on the right in Figure 4.2 indicates that this is correct.

Our main concern, however, is what happens if there is an error? That is simple to check as well. We can insert a random error for the function, here using the Matlab *rand* function to simulate a random error occurring in the computation of the function.

rand(1,1)-alphas(1)*sin(.35+1)-alphas(2)*sin(.35+2)-alphas(3)*sin(.35+3)-betao

Thus, returning a value of the checking polynomial

f(random) = 0.143

A test condition ceiling of $f(random) \geq 0.001$ would then clearly detect the error.

4.3.2 Matlab, Simulink, and DSP Builder

Matlab was chosen as the development environment for this algorithm for several reasons. Matlab (MATrix LABoratory) is a language designed for technical computing. It is designed to operate best on matrix and vector oriented data – which is the set of data for the numerical checking algorithm described above. It facilitates computation,

visualization, and programming in one environment. Familiar mathematical notation is standard throughout.

Matlab is becoming the mainstay for scientists, technicians, and engineers. It has a wealth of built-in subroutine libraries for analysis of fixed-point data. Many third-parties partner with Matlab or create toolboxes on their own to supplement the Matlab libraries.

In addition to a friendly and familiar environment Matlab also contains the simulation modeling package called Simulink. The Simulink package integrates with MATLAB for modeling, simulation, and analysis of dynamical systems in a graphical user interface (GUI) environment.

DSP Builder is an Altera product designed to be a Quartus II and Matlab/Simulink interface. Quartus II is a design environment for FPGA (field-programmable logic array), CPLD (complex programmable logic device), and structured ASIC (application specific integrated circuit) HardCopy Stratix device designs. DSP Builder automatically generates HDL code from a Simulink model, generating bit and cycle accurate models, automatically generating VHDL testbench data, and rapid prototyping with an Altera development board.

The drag-and-drop GUI Matlab/Simulink environment allows a designer to quickly create and deploy a system for testing and evaluation.

4.3.3 The m-file and GUI

The program for finding the coefficients of a checking polynomial is implemented as a Matlab function m-file. The function is called from the Matlab command line with the general statement

[k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN(‘s’,h,lower,upper,kk)

where the function inputs are defined as below in Table 4.4.

Table 4.4 Description of least square estimation function inputs.

s	Any valid Matlab function expression
h	Step size for interval
lower	Lower bound for test interval
upper	Upper bound for test interval
kk	Vector of shifted *k*'s to test

The function call in the example above

[k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN('sin(x)',0.001,0,6.283,(1:5))

will find the coefficients of the best checking polynomial for the sine function over the interval $x \in [0,2\pi]$ with a step size of $h = 0.001$ for $k \in [1,\ldots,5]$. In other words, it will test each of the checking polynomials

$$P_0(y_1,\ldots,y_k,f_0) = f_0^{d_0} + f_0^{d_0-1}A_{d_0-1}^0(y_1,\ldots,y_k) + \ldots + A_0^0(y_1,\ldots,y_k) = 0$$
$$P_1(y_1,\ldots,y_k,f_1) = f_1^{d_1} + f_1^{d_1-1}A_{d_1-1}^0(y_1,\ldots,y_k) + \ldots + A_0^1(y_1,\ldots,y_k) = 0$$
$$\vdots$$
$$P_5(y_1,\ldots,y_k,f_5) = f_5^{d_5} + f_5^{d_5-1}A_{d_5-1}^0(y_1,\ldots,y_k) + \ldots + A_0^5(y_1,\ldots,y_k) = 0$$

Type "help lsefunrun" at the command line for details on the function.

In addition to the command line function call, a graphical user interface (GUI) was developed that also allows a user to input any Matlab defined function, set the interval, and interval step size. The GUI then displays the required checking polynomial coefficients and the plots as described above.

4.3.4 Further examples

Here are three more examples of calls and results of the Matlab program.

First, consider the natural logarithm function. Note that the natural logarithm program is represented in Matlab as log(x). The function call to the Matlab prompt would be

[k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN('log(x)',0.001,1,7.283,(1:20))

This function call returns the output collected in Table 4.5 below.

Table 4.5 The least squares estimation function output for the natural logarithm function.

k	delta	alphas	betao	stepsize	A	B
15	6.275×10^{-5}	32.183 -232.022 560.872 -368.120 -170.815 10.918 103.354 168.655 87.432 -313.045 -18.425 71.158 369.342 -158.884 -145.765	19.148	0.001	1	7.283

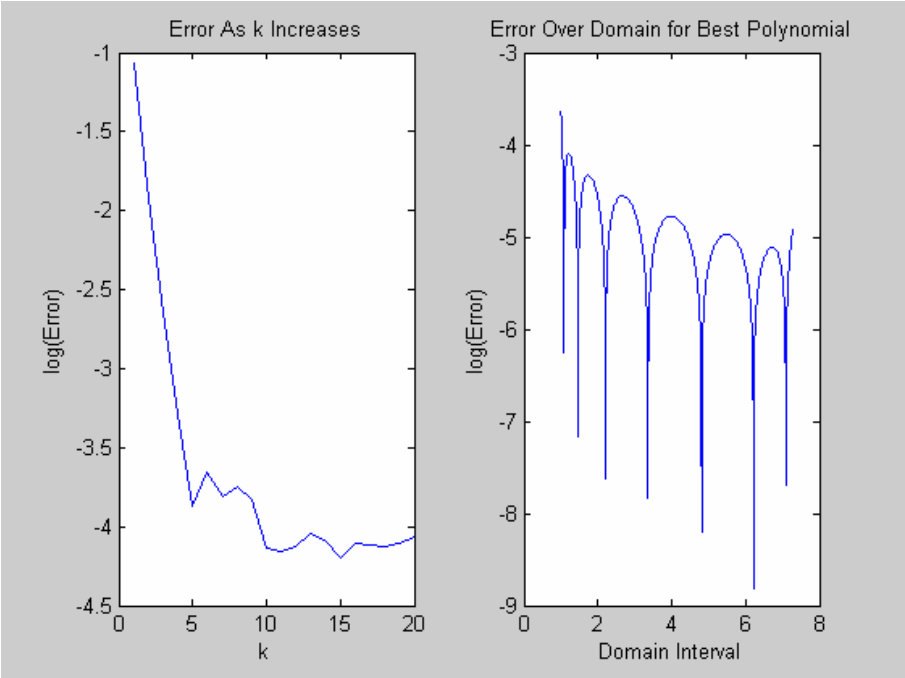


Figure 4.3 Plots of checking polynomial error for the natural logarithm function.

Comparing the error in the plots in Figure 4.3 to those shown for the sine function above we observe that the natural logarithm is not an LC function. This is indicated by its higher error values. However, we do see the same general behavior as *k* increases. The best checking polynomial has a much larger error than the sine function – on the order of 10^{-4} as compared to 10^{-15}. Since the function is not LC the checking polynomials do not reach minimum values until $k > 2$. In this case, we see minimization begin around $k = 5$. Absolute minimum for $k \in [1,\ldots,20]$ occurs at $k = 15$ resulting in the 15 *alphas* values in Table 4.4. Note also that the interval under test for this function is $x \in [0+1, 2\pi+1]$ because the logarithm is undefined at 0. Avoiding discontinuities is a problem that will be addressed below.

If we consider a test for $k \in [1,\ldots,40]$ the plot in Figure 4.4 is generated.

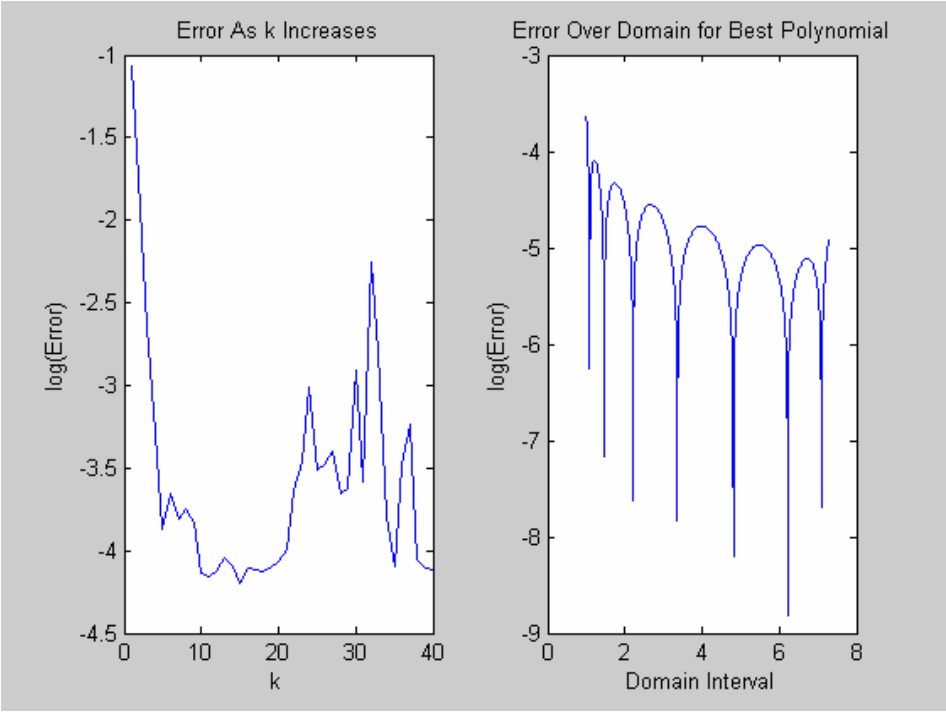


Figure 4.4 Checking polynomial error of the natural log function for $k \in [1,\ldots,40]$.

The plot shows that the minimum still occurs in this interval for k = 15. Notice after a certain point, around $k \in [20,\ldots,25]$, that the value of the delta function begins to increase. This is due to the increasing computational overhead of the algorithm. At around $k = 25$ the algorithm in fact breaks down due to the limitations of computer arithmetic. Recall from the least square estimation discussion above that the value of k indicates the dimension of the system to be solved.

For a final example, consider the more interesting function shown in Figure 4.5.

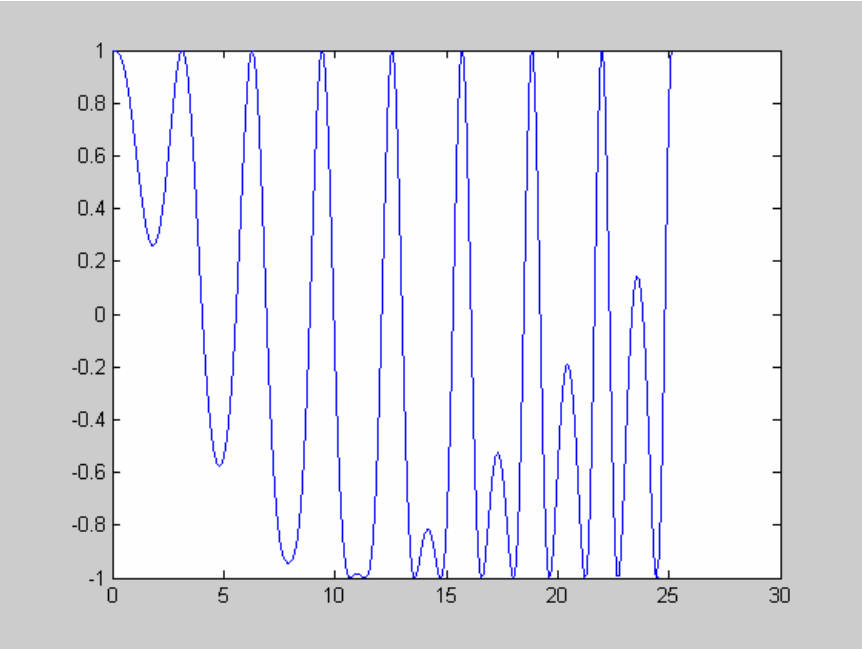


Figure 4.5 The function $f(x) = \cos\left(\sqrt{x}\sin(x)\right); x \in [0,4\cdot 2\pi]$.

Generate a checking polynomial for this function with the Matlab command line call

[k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN('cos(sqrt(x).*sin(x))',0.001,0,25.132,(1:70))

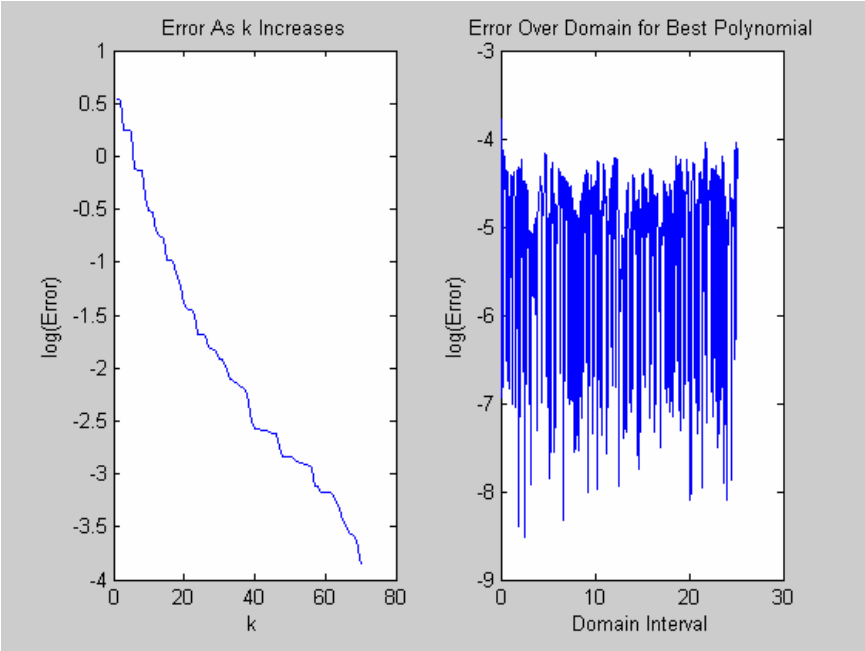


Figure 4.6 Plot of error on $f(x) = \cos\left(\sqrt{x}\sin(x)\right)$ for $k \in [1,\ldots,40]$.

Note that this function's checking polynomial was found up to $k = 70$ – meaning that a dimension 70 system was solved. The best polynomial was indeed the $k = 70$ checking polynomial. But notice that this is not an LC function. Thus, even with a very large checking polynomial, accuracy on the order of only 10^{-4} was achieved. However, we shouldn't dismiss the potential of this checking polynomial. A design with sufficient parallelism or pipelining could take advantage of even this high-order checking polynomial.

4.3.5 Code conversion

Matlab provides support for converting MATLAB applications to C and C++ code, Excel add-ins, and COM components as well as accessing data contained in databases, other applications, and from instruments. Thus, the LSEFUNRUN function used to generate coefficients of checking polynomials for this work could be further developed or applied in many directions.

Chapter 5

A Short Library of Checking Polynomials

5.1 Choosing appropriate parameters

A designer who seeks to employ a checking polynomial must first choose appropriate values for the function described in this paper that will generate the coefficients of the checking polynomial. The generating Matlab function, LSEFUNRUN, requires the following parameters to be set: *stepsize*, lower domain limit *A*, and upper domain limit *B*. When setting these parameters the designer should keep in mind what other system constraints might bear on this decision. Other system constraints that could have an impact include instrument accuracy, D/A and A/D issues, system bit-size, and time or space constraints.

For the purpose here of creating a sample of checking polynomials the default Matlab/Simulink block parameters will dictate the choice. This will allow for consistency with the implementation discussion below. Simulink look-up table functional blocks have a default accuracy of 10^{-5} for fixed-point computations. For the simplicity of this demonstration we will define $h = 10^{-4}$. In determining the domain interval a suitable interval will be chosen based on the function's periodicity and continuity.

As mentioned above a best *k* value must also be chosen based on design objectives. This will be the smallest *k* for which $\delta_k \leq 10^{-5}$.

5.2 Common numerical functions

Some of the coefficients shown in table 5.1 below are included in exponential format for academic purposes. Rounding would occur as appropriate. All values are truncated at five decimal places.

5.2.1 Simple functions

Table 5.1 A short collection of simple functions.

Function	Interval	Best k	δ	$\alpha_1,\ldots,\alpha_k$	β_o
$\sin(x)$	$[0,\pi]$	2	1.01418×10^{-13}	1.08060 -0.99999	1.01188×10^{-13}
$\cos(x)$	$[0,\pi]$	2	3.82181×10^{-14}	1.08060 -1.00000	-1.76920×10^{-14}
$\tan(x)$	$[0,\frac{\pi}{4}]$	7	2.66824×10^{-5}	9.81682×10^{-11} 0.06606 2.70513 -3.99153×10^{-11} -0.01866 -1.76710 0.00015	-0.04742
e^x	[0,1]	1	4.07913×10^{-14}	0.36787	1.26019×10^{-13}
e^{-x}	[0,1]	1	4.53024×10^{-14}	2.71828	-1.61613×10^{-13}
$\ln(x)$	[1,2]	5	1.99536×10^{-5}	24.25068 -112.20551 145.22812 -4.68669 -54.57269	10.45630
$\sqrt{x}$	[1,2]	3	8.09569×10^{-6}	11.31481 -26.72049 16.84968	-2.41964
x^2	[0,10]	2	1.78242×10^{-10}	2.00000 -1.00000	2.00000
$\sinh(x)$	$[-5,5]$	2	1.65782×10^{-11}	3.08616 -0.99999	-3.54325×10^{-12}
$\operatorname{arccot}(x)$	$[0.01,3.14]$	5	0.000410024	53.76578 -669.00109 2777.03653 -4413.65061 2332.73726	-3.21939

5.2.2 Compound functions

Table 5.2 below shows some results for functions that are a combination of simple numerical functions. The coefficients of their checking polynomials are again determined as outlined above.

Table 5.2 A short collection of compound numerical functions.

Function	k	δ	$\alpha_1,\ldots,\alpha_k$	β_o
$e^{-x}\sin(x)$	2	1.12254×10^{-14}	2.93738 -7.38905	-6.25552×10^{-15}
$e^{x}\sin(x)$	2	6.18880×10^{-14}	0.39753 -0.13533	4.81428×10^{-14}
$\sqrt{x}\sin(x)$	8	0.00052571	93.39562 -447.62812 720.26276 -224.71791 -967.50367 1808.29719 -1392.66260 559.86044	-1.96677
$\frac{x^2\sin(x)}{\sqrt{x}}$	7	0.00018150	5.01346 -11.29504 22.87906 -28.02754 27.52611 -15.15646 5.91847	-0.77675
$\cos^2\left(\sqrt{x}\right)+\sin\left(\sqrt{x}\right)-e^{-\sin(x)}$	13	0.000126603	275.76566 35.34309 7.18978 24.70067 75.54123 -28.28778 -578.92236 -81.77037 -43.48980 -57.96112 -96.76949 -60.67816 423.95965	259.58941

Chapter 6

Application of a Checking Polynomial

6.1 Hardware implementation

Checking polynomials in hardware could be pipelined or parallel implementations. In a pipelined implementation the computation of the function is stored in system memory while the values of *f* at points $x, x+a_1, \ldots, x+a_k$ accumulate in a buffer. Once the *k*th value is computed those values are sent to a combinational block. The combinational block then computes $P\left(\tilde{f}_0, \ldots, \tilde{f}_k\right)$, the absolute value of which is compared to δ for the error check. If *P* is less than or equal to δ then the computation is fault free. If *P* is greater than δ then a fault is detected.

6.1.1 Example Simulink designs

As mentioned above the Matlab/Simulink environment will be used for demonstrating the deployment of a checking polynomial to hardware. First, consider the implementation of a sine function checker. Simulink contains a sine look-up table for $f(x) = \sin(2\pi x)$. A run of LSEFUNRUN for this function gives the results shown below in Table 6.1.

Table 6.1 Program output data for $f(x) = \sin(2\pi x)$.

k	1
delta	5.27966×10^{-16}
alphas	1
betao	-6.36028×10^{-17}
stepsize	0.0001
A	0
B	1

The fact that this is an LC function makes its implementation simple. A Simulink model was designed that implements this checking polynomial. The systems below are not meant to be practical designs but as exhibits of how a checking polynomial functions. Practical implementations would likely involve the creation of a user-defined functional block that contained the checking polynomial "wrapper" hardware. With such a user-defined function a designer could then simply drag-and-drop a fault-secure block anywhere in the design.

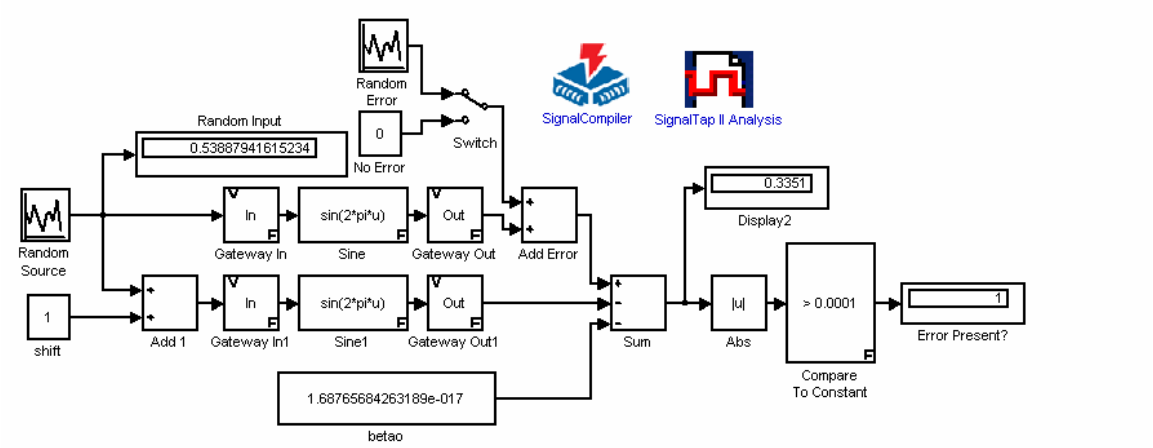


Figure 6.1 Simulink implementation of checking algorithm for $f(x) = \sin(2\pi x)$.

The Simulink model in Figure 6.1 evaluates the function $f(x) = \sin(2\pi x)$ for some random input $x \in [0,1]$. A switch is included to allow a random value to be added to the computation of the sine function. The state shows in Figure 6.1 that it has detected the random error insertion – by an active 1 indication on the output. This event could then trigger the error location and correcting described above in the example of section 3.1. The constant term *betao* is included for illustrative purposes but doesn't add to the practical function of this system because it is so small. Although the sine function is shown in two separate modules in the diagram it can be thought of as one look-up table

module returning $f(x)$ and $f(x+1)$ in parallel. As mentioned above, the system could be designed to pipeline this process by storing the previously computed value.

Another example is shown below in Figure 6.2 for $f(x) = e^x \sin(x)$, a compound function as computed in Table 5.2 above.

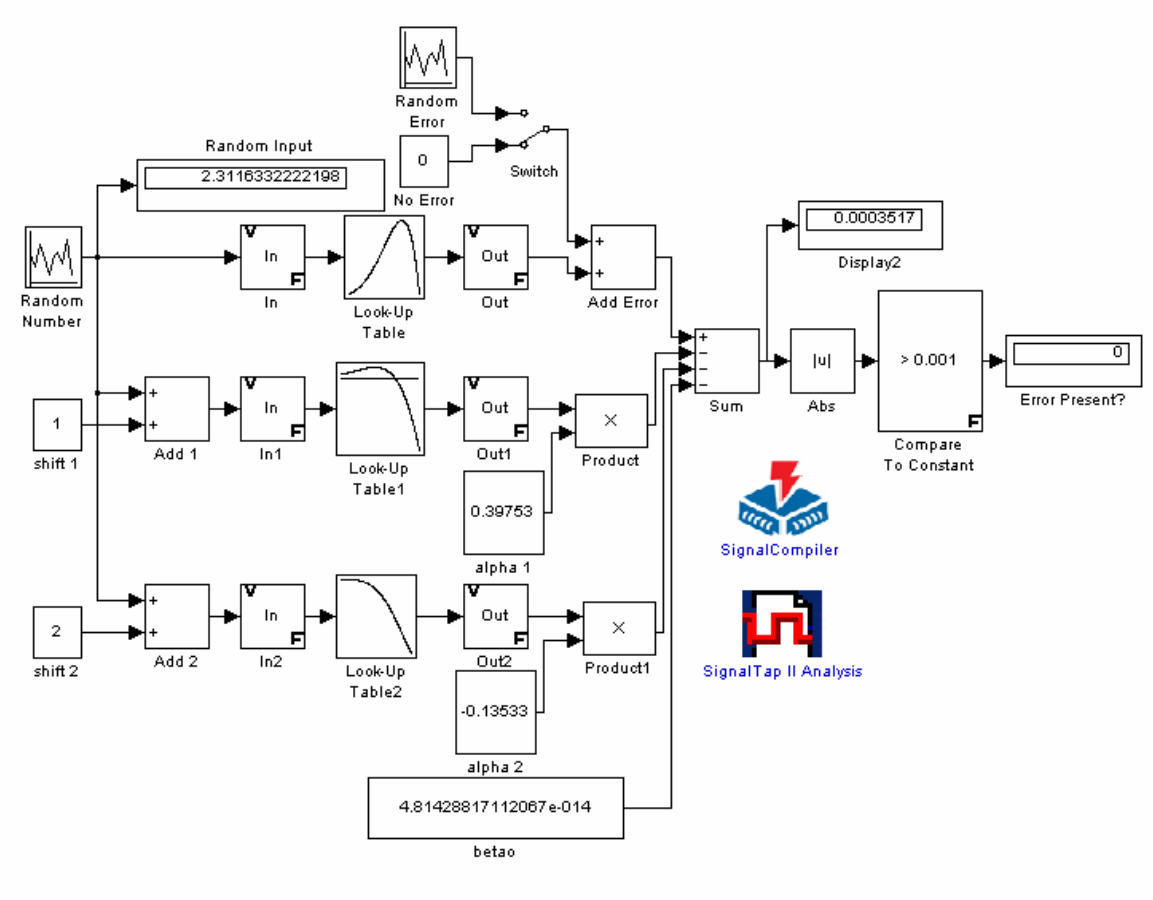


Figure 6.2 Simulink implementation of checking algorithm for $f(x) = e^x \sin(x)$.

As in Figure 6.1, the look-up table is shown in Figure 6.2 in three separate modules. This could in reality be one look-up table module that returned the function value and two shifted function values in parallel.

6.1.2 FPGA Deployment

The Signal Compiler and Signal Tap II Analysis features shown in the Simulink implementations of Figures 6.1 and 6.2 can convert the model to VHDL for simulation in the Quartus II development environment and optional deployment to the FPGA Stratix family target boards. Similar products exist from other FPGA vendors such as Xilinx,. FPGA technology is mentioned here because it has improved to the point where it is attractive for safety and mission-critical applications. NASA is among those considering the use of commercial FPGA solutions as a cost-saving measure for deep space exploration over high-cost radiation shielding [Turmon 03].

6.2 Hardware overhead

The hardware (or software) implementation of a checking polynomial can take many forms. Typical hardware elements could include the following for a *b*-bit, *m*-order checking polynomial:

Counter: $k+1$ flip flops = $8(k+1)$ gates.

Incrementor: $5m$ gates per counter $\rightarrow 5mk$ gates.

Buffer: $(k+1)b$ flip flops $= 8(k+1)b$ gates.

Combinational block: 1 multipler: $8b^2$ gates; 1 adder $8b$ gates $\rightarrow 8b+8b^2$ gates.

Comparison block: $3b$ gates.

Sample total configuration: $8b^2+11b+8(k+1)(b+1)+5mk$ gates.

For example, the space complexity of ROM with $m=16, b=32$ is $L \approx 14b \cdot 2^m \approx 3 \cdot 10^7$ gates. The space complexity of overhead then is $L_k = 8808 + 344k$ and

$$\frac{L_k}{L} \approx \frac{8808 + 344k}{3} \cdot 10^{-7}$$

Thus, the space complexity of the overhead is very small – even for large *k*.

6.3 Fault coverage

For a *b*-bit ROM suppose function $f(x)$ is normalized such that $f(x) \le 1 - 2^{-b}$ and the number *y* is given as a string $y = sy_1y_2 \ldots y_b$, where *s* represents the sign of *y* and y_b is the least significant bit. In other words, $y = sign\left(2^{-1}y_1 + 2^{-2}y_2 + \cdots + 2^{-b}y_b\right)$. If the formula

$$\left| \sum_{i=0}^{k} c_i \tilde{f}_i \right| \le \delta \,;\; c_0 = 1 \tag{7}$$

is used then the value of δ depends on the function *f* and on the interval where it is computed. It was shown by [Vainstein 91] that in the case where the calculation is correct

$$\delta = \sum_{i=0}^{k} |c_i| \cdot 2^{-b} \tag{8}$$

Using (8) it can be shown that the portion of undetected errors is smaller or equal to $\frac{\delta}{\max_i C_i}$ for uniformly distributed random variable X_i and $f_{X_i}(0) = \frac{1}{2C_i}$, $i = 0, \ldots, k$.

Example: We considered $f(x) = \sin(x)$ above and found the checking polynomial for $a_1 = 1, a_2 = 2$ to be $f_0 - \frac{\sin(2)}{\sin(1)} f_1 + f_2 = 0$, and used $\left| \tilde{f}_0 - \frac{\sin(2)}{\sin(1)} \tilde{f}_1 + \tilde{f}_2 \right| \le \delta$ for error detection. Using (8) then $\delta = 2^{-b}\left(1 + \frac{\sin 2}{\sin 1} + 1\right) < 3.1 \cdot 2^{-b}$. The percentage of detected

faults then is greater or equal to $d(b) = \left(1 - \frac{2^{-b}\left(1 + \frac{\sin 2}{\sin 1} + 1\right)}{\frac{\sin 2}{\sin 1}}\right) \cdot 100\%$. In the case of a 32-bit ROM this gives a percentage of errors detected of $d(32) = 99.999999933624\%$.

6.4 Markets

The application of checking polynomials, especially in the area of numerically intensive applications, gives the system designer greater flexibility and choice when deciding between commercial off-the-shelf (COTS) FPGA or similar technologies and the more expensive environmentally hardened processors.

Recent efforts in the application of algorithm based fault tolerance (ABFT) [Katz 99, Turmon 00-03] to numerically intensive, remote, high-performance space applications indicate that the implementation of checking polynomials can serve as a viable option. COTS FPGA technology with secure fault tolerant numerical calculations using checking polynomials saves design space and time. This approach can be a step towards saving the expense of shielding the processor from damaging environmental interference such as radiation.

Plans are currently underway to deploy and test checking polynomials to numerically intensive applications in FPGA based devices.

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

This thesis considered checking polynomials as a method for achieving fault tolerant numerical computations. A program was presented which computes the coefficients of a checking polynomial for a broad class of numerical functions. A short library of checking polynomials for common numerical functions and examples of checking polynomials for compound numerical functions were listed. Significantly, a Matlab function is provided that will allow a developer to determine a checking polynomial for many numerical functions.

Functional models were given of checking polynomials integrated into numerical functions. Options for deploying these models to FPGA or similar hardware test beds were given. Results in [Dido 02] indicate that complex numerical algorithms and floating point units may be implemented in an FPGA.

Although a hardware implementation is presented here it should be noted that a software implementation is equally viable and may be more appropriate in applications that are not safety or mission critical.

With the program and methods made available in this thesis the system designer can quickly find and use a checking polynomial as a method for fault tolerant computation of numerical functions.

7.2 Future work

Although the set of functions covered by this work is broad there remain problems among subsets of the set of numerical functions which should be considered. Of particular interest will be functions over intervals containing discontinuities and how they can be tamed. Also to be considered are functions of several variables, partially polynomially checkable (PPC) functions, and error detecting for faults of different multiplicity.

A Matlab toolbox is a package of related functions that can foster investigation and work in a particular area. The function developed or this thesis serves the purpose of finding coefficients of checking polynomials. With the investigation of the other types of checking polynomials mentioned above a Matlab or C toolbox could be developed that contains functions related to finding checking polynomials for other types of functions. Such a Matlab toolbox could be designed to smoothly integrate with other Matlab functions used in guidance, navigation, and control design allowing for “drag-and-drop” of fault-secure numerical functions into a design.

Further work could consider specific processor or function implementations. The rapid growth of system-on-chip (SOC) complexity has created a need for space-sensitive fault tolerant strategies to replace the traditional strategy of processor module replication with voting – a strategy that can consume a great deal of silicon real estate. This need is especially urgent for DSP SOC processors with intensive numerical algorithms. As another example, nanosystems could benefit from the very low overhead of this fault tolerant numerical function technique by freeing up system resources currently devoted to space, time, or code redundancy techniques.

Appendix A

Matlab Code for LSEFUNRUN Function

```
function [k,delta,alphas,betao,stepsize,A,B]=lsefunrun(s, h, lower, upper, kk)
%   LSEFUNRUN Least Square Estimation Function
%   [k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN(s,h,lower,upper,kk) returns
alphas, betao, and delta
%
%   k = the value of k from those tested returning the best estimation
%   delta = the delta approximation
%
%   s = the expression to test. Enter as 'sin(x)' Use matlab '.' operator as required
%       enter "help arith" on the Matlab command line for more info
%   stepsize = h = distance between domain elements
%   A = lower = the lower limit of interval
%   B = upper = the upper limit of interval
%   kk = vector containing the number of a's (shifted functions) to try in this run
%       for example: kk=(1:10) will run the test for k=1,2,3,...,10.
%
%   This version will produce a graph of delta versus k and return the value of k that
%   gives the minimum delta approximation
%
%   For example, to run the test for the natural logarithm function (log(x)
%   in Matlab) for domain step size h=0.01, for x in [1,2], for number of
%   shifted functions in the range [1:10] then enter
%       [k,delta,alphas,betao,stepsize,A,B]=LSEFUNRUN('log(x)',0.01,1,2,(1:10))

t = 1e-30;  % This sets the tolerance for rref rank tests
%tic;
format long g;

%x=(lower:h:upper)';        % x = Values in the domain of f
%y=eval(s);            % y = Values in the range of f
%Y=[x y];              % Express f as a table with domain in col1 and range in col2

% Initialize deltas vector
deltas=[];
allalphas=[];
allbetao=[];

% Now extend Y so that
%       column 3 -> f1
%       column 4 -> f2
%       ...
%       column k+2 -> fk
```

```
% Expand matrix Y

%Y(1:upper,3:(k+2))=0

% Need k columns and B rows
% note that A(:,j) is the jth column of A

z=length(kk);

while (z>0)

x=(lower:h:upper).';       % x = Values in the domain of f
y=eval(s);           % y = Values in the range of f
Y=[x y];

k=kk(length(kk)-z+1);
aa=(1:k);

for n = 1:k
   % find the shifted domain values
   x = x+aa(n);
   % put the values of the shifted functions in the last columns of Y
   temp=eval(s);
   Y=[Y temp];
   %reset x to the original for new shift
   x=(lower:h:upper).';
end

Y;

% Define A to be the matrix of integral products <f,g>=trapz(f .* g)
%              using the trapezoidal integration method
% where the matrix has this layout:
%
%      <1,f1>  ... <1,fk>  (B-A)
%      <f1,f1> ... <f1,fk> <f1,1>
% A =   <f2,f1> ... <f2,fk> <f2,1>
%      ...
%      <fk,f1> ... <fk,fk> <fk,1>
%
% This will be matrix A in the matrix equation A*X=B
% solved by X=B\A in Matlab
%
% Matrix X will be a column matrix of
% X = [ alpha1 alpha2 ... alphak betao ]'
```

```
%
% and Matrix B will be a column matrix of the integral products
% B = [ <fo,1> <fo,f1> <fo,f2> ... <fo,fk> ]'
%

% Create matrix A
A = zeros(k+1);

% Fill in matrix A with the appropriate integral products
% First, fill in the first row except last entry of A
% This is <1,f1>, <1,f2>, ..., <1,fk>
for n = 1:k
   A(1,n)= trapz(x,ones(((upper-lower)/h)+1,1) .* Y(:,n+2)); % These are the <1,fk>
items
end
x;
A;

% Second, fill in the last entry of the first row of A
% this is the (B-A) entry in top right corner of matrix
A(1,k+1)=(upper-lower);

A;

% Third, fill in 2nd thru kth rows except last column of A
for n=2:k+1          % Cover Rows
   for m=1:k       % Cover Columns
      A(n,m) = trapz(x,Y(:,n+1) .* Y(:,m+2));
   end
end

A;

% Finally, fill in the last column of A
for n=2:k+1
   A(n,k+1) = trapz(x,ones(((upper-lower)/h)+1,1) .* Y(:,n+1));
end

%A

% Create the column vector B
B=zeros(k+1,1);

%B;

% And populate with appropriate integral products
```

```
B(1,1) = trapz(x,ones(((upper-lower)/h)+1,1) .* Y(:,2));
for n=1:k
   B(n+1,1) = trapz(x,Y(:,2) .* Y(:,n+2));
end

%B

% Solve the equation A*X=B
%I=inv(A);

% X contains alpha1, alpha2, ... alphak, betao
%X=I*B;
%X=(B\A)';

%X

% Solve the matrix equation by using the rref form

M=[A B];

%Use a very small tolerance
R=rref(M,t);

% Assign X to be the last column of R
X=R(:,k+2);

% Evaluate the delta function of betao, alpha1, ...

% First, form the terms of the polynomial in the delta function
% these are: fo, alpha1*f1, alpha2*f2, ..., alphak*fk
%
% where delta[2]=alpha1, delta[3]=alpha2, ... delta[k+1]=alphak
%       delta[1]=betao
%
% and   Y(:,2)=f, Y(:,3)=f1, Y(:,4)=f2, ..., Y(:,k+2)=fk

integrand=Y(:,2);    % Initialize integrand to f

%integrand
%X(1)
%Y(:,1+2)
%temp = X(n)*Y(:,1+2)
%integrand-temp

for n=1:k
   %X(n)
```

```
   %X(n)*Y(:,n+2)
   %integrand-(X(n)*Y(:,n+2))
   integrand = integrand - ( X(n) * Y(:,n+2) ); % form polynomial
end

%integrand

integrand = integrand - X(k+1); % subtract betao

%integrand

% Now square the polynomial
integrand = integrand.*integrand;

% Integrate to get value of delta function by trapezoidal method

delta=trapz(x,integrand);
alphas=X(1:k);
betao=X(k+1);

%Create the vector of deltas
deltas=[deltas; delta];

allalphas=[allalphas; alphas];
allbetao=[allbetao; betao];

z=z-1;

end     % While Loop END

% Print a Table of ks and deltas and make a plot
expression = s;
%toc;
Table=[kk' deltas];

%    [Y,I] = MIN(X) returns the indices of the minimum values in vector I.
%    If the values along the first non-singleton dimension contain more
%    than one minimal element, the index of the first one is returned.
%    [Y,I] = MIN(X,[],DIM) operates along the dimension DIM.

[Y,I]=min(Table(:,2));

k=I(1);
% This is the minimum delta value
delta=sqrt(abs(Table(I(1),2)));
```

```
% These are the alphas and betao for the k that returns minimum delta
sum=0;
for n=1:(k-1)
   sum=sum+n;
end
alphas = allalphas(sum+1:sum+k);
betao = allbetao(k);

% These are the parameters of the run
stepsize=h;
A=lower;
B=upper;

% Code added to plot in separate window
figure(1);

subplot(1,2,1);
plot(kk,log10(sqrt(abs(deltas))));
xlabel('k');
ylabel('log(Error)');
title('Error As k Increases');

% find f0-alpha1*f1-...-betao
x = lower:h:upper;
f0 = eval(s);
for i=1:length(alphas)
   x = (lower + i):h:(upper+i);
   fi = eval(s);
   f0 = f0 - alphas(i)*fi;
end
f0 = f0 - betao;

x = lower:h:upper;

subplot(1,2,2);
plot(x,log10(abs(f0)));
xlabel('Domain Interval');
ylabel('log(Error)');
title('Error Over Domain for Best Polynomial');
```

Appendix B

Matlab Code for LSE Graphical User Interface

```
function varargout = LSE(varargin)
% LSE Application M-file for LSE.fig
%    FIG = LSE launch LSE GUI.
%    LSE('callback_name', ...) invoke the named callback.

% Last Modified by GUIDE v2.0 04-Dec-2002 09:39:15
tic;

if nargin == 0  % LAUNCH GUI

        fig = openfig(mfilename,'reuse');


        % Generate a structure of handles to pass to callbacks, and store it.
        handles = guihandles(fig);
        guidata(fig, handles);

        if nargout > 0
                varargout{1} = fig;
        end

   set(handles.UserDefined, 'Value', 1);
   setUserDefinedGUI(handles)

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK

        try
                if (nargout)
                        [varargout{1:nargout}] = feval(varargin{:}); % FEVAL
switchyard
                else
                        feval(varargin{:}); % FEVAL switchyard
                end
        catch
                disp(lasterr);
        end

end

% --------------------------------------------------------------------
function varargout = Run_Callback(h, eventdata, handles, varargin)
```

```
option =  get(handles.UserDefined,'Value') ;

if (isempty(get(handles.Expression, 'string' )) | ...
   isempty(get(handles.Distance, 'string' )) | ...
   isempty(get(handles.LowerLimit, 'string' )) | ...
   isempty(get(handles.UpperLimit, 'string' )))

   errordlg('An undefined parameter located');
else

   if option == 1 % UserDefined

      if (isempty(get(handles.NumberA, 'string' )) | ...
         isempty(get(handles.VectorShiftA, 'string' ))| ...
         isempty(get(handles.Tolerance, 'string' )))

         errordlg('An undefined parameter located.');
      else

         s = strrep(get(handles.Expression, 'string'), "","");
         h = str2double(get(handles.Distance, 'string'));

         lower =  str2num(get(handles.LowerLimit, 'string'));
         upper =  str2num(get(handles.UpperLimit, 'string'));
         k = str2num(get(handles.NumberA, 'string'));
         aa = eval(get(handles.VectorShiftA, 'string'));

         if( length(aa) ~= k )

            errordlg('k must match size of shifting a vector');

         else

            t = str2double(get(handles.Tolerance, 'string'));

            [a, b, d] = lsefunx(s, h, lower, upper, k, aa, t);

            set(handles.A_Coefficients, 'String', a);
            set(handles.Future_k_Coefficient, 'String', b);
            set(handles.Delta_Integral, 'String', d);

         end
      end
   else % option = 0 Approximation

      if isempty(get(handles.VectorA, 'string' ))
```

```
            errordlg('An undefined parameter located');
        else

            s = strrep(get(handles.Expression, 'string'), "","");
            h = str2double(get(handles.Distance, 'string'));
            lower =  str2num(get(handles.LowerLimit, 'string'));
            upper =  str2num(get(handles.UpperLimit, 'string'));
            kk = eval(get(handles.VectorA, 'string'));

            [k, app] = lsefunrun(s, h, lower, upper, kk);

                set(handles.Expression_out, 'String', s);
                set(handles.Elapsed_Time, 'String', toc);
                set(handles.kValue, 'String', k);
                set(handles.Delta_Approx, 'String', app);
            end
        end

    end

% --------------------------------------------------------------------
function varargout = Approximation_Callback(h, eventdata, handles, varargin)

    off = [handles.UserDefined];
    mutual_exclude_off(off);

    on = [handles.Approximation];
    mutual_exclude_on(on);

    enabledC = [.693,.746, .783];
    disabledC =  [.831,.816, .784];

    % Now disable all the text boxes to not allow user input / system output
    set(handles.NumberA,'Enable','off');
    set(handles.NumberA,'BackgroundColor', disabledC);

    set(handles.VectorShiftA,'Enable','off');
    set(handles.VectorShiftA,'BackgroundColor', disabledC);

    set(handles.Tolerance,'Enable','off');
    set(handles.Tolerance,'BackgroundColor', disabledC);

    set(handles.A_Coefficients,'Enable','off');
```

```
    set(handles.A_Coefficients,'BackgroundColor', disabledC);

    set(handles.Future_k_Coefficient,'Enable','off');
    set(handles.Future_k_Coefficient,'BackgroundColor', disabledC);

    set(handles.Delta_Integral,'Enable','off');
    set(handles.Delta_Integral,'BackgroundColor', disabledC);

    % Now enable all the text boxes to allow user input
    set(handles.VectorA,'Enable','on');
    set(handles.VectorA,'BackgroundColor', enabledC);

    set(handles.Expression_out,'Enable','inactive');
    set(handles.Expression_out,'BackgroundColor', enabledC);

    set(handles.Elapsed_Time,'Enable','inactive');
    set(handles.Elapsed_Time,'BackgroundColor', enabledC);

    set(handles.kValue,'Enable','inactive');
    set(handles.kValue,'BackgroundColor', enabledC);

    set(handles.Delta_Approx,'Enable','inactive');
    set(handles.Delta_Approx,'BackgroundColor', enabledC);



% --------------------------------------------------------------------
function varargout = UserDefined_Callback(h, eventdata, handles, varargin)


    off = [handles.Approximation];
    mutual_exclude_off(off);

    on = [handles.UserDefined];
    mutual_exclude_on(on);

    setUserDefinedGUI(handles)

% --------------------------------------------------------------------
function mutual_exclude_off(off)

    set(off,'Value',0)

% --------------------------------------------------------------------
function mutual_exclude_on(on)
```

```
    set(on,'Value',1)

% --------------------------------------------------------------------
function setUserDefinedGUI(handles)

    enabledC = [.693,.746, .783];
    disabledC =  [.831,.816, .784];

     % Now disable all the text boxes to not allow user input / system output
    set(handles.VectorA,'Enable','off');
    set(handles.VectorA,'BackgroundColor', disabledC);

    set(handles.Expression_out,'Enable','off');
    set(handles.Expression_out,'BackgroundColor', disabledC);

    set(handles.Elapsed_Time,'Enable','off');
    set(handles.Elapsed_Time,'BackgroundColor', disabledC);

    set(handles.kValue,'Enable','off');
    set(handles.kValue,'BackgroundColor', disabledC);

    set(handles.Delta_Approx,'Enable','off');
    set(handles.Delta_Approx,'BackgroundColor', disabledC);

     % Now enable all the text boxes to allow user input / system output
    set(handles.NumberA,'Enable','on');
    set(handles.NumberA,'BackgroundColor', enabledC);

    set(handles.VectorShiftA,'Enable','on');
    set(handles.VectorShiftA,'BackgroundColor', enabledC);

    set(handles.Tolerance,'Enable','on');
    set(handles.Tolerance,'BackgroundColor', enabledC);

    set(handles.A_Coefficients,'Enable','on');
    set(handles.A_Coefficients,'BackgroundColor', enabledC);

    set(handles.Future_k_Coefficient,'Enable','inactive');
    set(handles.Future_k_Coefficient,'BackgroundColor', enabledC);

    set(handles.Delta_Integral,'Enable','inactive');
    set(handles.Delta_Integral,'BackgroundColor', enabledC);


% --------------------------------------------------------------------
function varargout = Expression_Callback(h, eventdata, handles, varargin)
```

```
% --------------------------------------------------------------------
function varargout = Distance_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = LowerLimit_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = UpperLimit_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = NumberA_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = VectorA_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = VectorShiftA_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = Tolerance_Callback(h, eventdata, handles, varargin)

% --------------------------------------------------------------------
function varargout = A_Coefficients_Callback(h, eventdata, handles, varargin)
```

References

[Alefeld 83] Alefeld and Herzberger, Introduction to Interval Computation, Academic Press, New York, 1983

[Asumth 82] C. A. Asumth and G. R. Blakley, Polling splitting and restituting information to overcome total failure of some channels of communication, Proc. of the 1982 Symposium on Security and Privacy, IEEE Society, pp. 156-169.

[Blakely 85] G. R. Blakely and C. Meadows, Security of Ramp Schemes, pp. 242-268 in the book Advances in Cryptology: Proceedings of Cripto 84, Edited by G. R. Blakley and David Chaum (Vol. 196 of Lecture Notes in Computer Science) Springer-Verlag. Berlin, 1985.

[Blum 88] M. Blum. Designing Programs to Check their Work. ICSI TR-88-009, 1988.

[Blum 89] M. Blum, S. Kannan. Designing Programs that Check their Work. In Proc. 21st ACM Symposium on Theory of Computing, 1989.

[Blum 90] M. Blum, M. Luby, R. Rubinfeld, Self-Testing/Correcting with Applications to Numerical Problems, Proc. 22nd ACM Symposium on Theory of Computing, 1990.

[Blum 90] M. Blum, M. Luby and R. Rubinfeld, Self-Testing and Self-Correcting Programs with Applications to Numerical Problems, 22nd ACM Symposium on Theory of Computing, pp. 73-83, 1990.

[Blum 95] M. Blum, B Codenotti, P. Gemmell, T. Shahoumian. Self-Correcting for Function Fields of Finite Transcendental Degree. ICALP: Annual International Colloquium on Automata, Languages and Programming, 1995.

[Dido 02] J. Dido, et. al., A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA based DSPs, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, pp. 50-55, 2002.

[Esonu 94] M.O. Esonu, A.J. A1-Kalili, S. Hariri, D. A1-Kalili, Fault tolerant design methodology for systolic array architectures, IEE Proceedings E 141 (1) (1994) 17-28.

[Karpovsky 79] M. G. Karpovsky, Error Detection for Polynomial Computations. IEEE J. Computing & Digital Tech., 2, (1), pp. 49-56, 1979.

[Karpovsky 80] M. G. Karpovsky, Testing for Numerical Computations, IEEE Proc. E. Computing & Digital Tech., 127, (2), pp. 69-76, 1980.

[Karpovsky 82] M. G. Karpovsky, Detection and Location of errors by Linear Inequality Checks. IEEE Proc., Vol. 129, Pt. E, No. 3, 1982..

[Katz 99] D. S. Katz, P. Springer, and M. Turmon, Software Fault Tolerance for Remote Exploration and Experimentation, High-Performance Embedded Computing Workshop, MIT Lincoln Laboratories, 1999

[Lang 65] S. Lang. Algebra. Addison-Wesley Publishing Co., 1965.

[Lipton 91] R. Lipton. New directions in testing, in Distributed Computing and Cryptography, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 2, American Mathematical Society, 1991.

[Malcoci 98] A. Malcoci, W. Jacquet, S. Breton, J.-P. Urban. Growing Hyperplan-Based Self-Organizing Maps for Function Approximation. Symposium of Electronics and Telecommunications. September 1998.

[Moore 79] R. E. Moore, Methods and Applications of Interval Analysis, SIAM, Philadelphia, 1979.

[Neumaier 90] A. Neumaier, Interval Methods for Systems of Equations, Cambridge University Press, 1990.

[von Neumann 56] von Neumann, J. Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components, Automata Studies, Princeton University Press, 1956.

[NIST 95] National Institute of Standards and Technology. A Conceptual Framework for System Fault Tolerance. 1995.

[Parag 85] Parag K. Lala. Fault Tolerant & Fault Testable Hardware Design. Prentice-Hall International. London 1985

[Siewiorek 82] Daniel P. Siewiorek and Robert S. Swarz, The Theory and Practice of Reliable System Design, Digital Press, 1982.

[Starzyk 01] J. A. Starzyk and Y. Guo, Reconfigurable Self-Organizing NN Design Using Virtex FPGA, Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA) Las Vegas, NV, June 2001.

[Turmon 00] M. Turmon and R. Granat, Algorithm-Based Fault Tolerance for Spaceborne Computing: Basis and Implementations, Proc. IEEE Aerospace Conference, vol. 4, 411-420, 2000.

[Turmon 00] M. Turmon, R. Granat, and D. S. Katz. Software-Implemented Fault Detection for High-Performance Space Applications, Proc. IEEE Conf. Dependable Systems and Networks, 2000, 107-116.

[Turmon 03] M. Turmon, R. Granat, D. Katz, and J. Z. Lou, Tests and tolerances for High-Performance Software-Implemented Fault Detection, in IEEE Trans. Computers, May, 2003.

[Vainstein 91] F. Vainstein. Error Detection and Correction in Numerical Computation by Algebraic Methods. Lecture Notes in Computer Science, 539, Springer Verlag, 1991, pp.456-464.

[Vainstein 93] F. Vainstein. Algebraic Methods in Hardware/Software Testing. Ph.D. Thesis, Boston University Graduate School, 1993.

[Vainstein 95] F. S. Vainstein, I. A. Kateeb Testing of Computations by Polynomial Checks, Proceedings of 27th IEEE Southeastern Symposium on System Theory, pp. 414-416, 1995.

[Vainstein 96] F. S. Vainstein, Low Redundancy Polynomial Checks for Numerical Computations, Applicable Algebra in Engineering, Communication and Computing, vol. 7, No. 6, pp. 439-447, 1996.

[Vainstein 98] F. S. Vainstein Self-Checking Design Procedure for Numerical Computations, VLSI Design, vol. 5, No. 4, pp. 385-392, 1998.

[Vainstein 01] F. S. Vainstein, Testing Programs Computing Numerical Functions, The 12th International Symposium on Software Reliability Engineering (ISSRE 2001), pp. 23-24, 2001.

[Vijay 97] M. Vijay and R. Mittal, Algorithm Based Fault Tolerance: a Review. Microprocessors and Microsystems 21 (1997) 151-161.

[Wasserman 97] H. Wasserman, M. Blum. Software Reliability via Run-Time Result-Checking. Journal of the ACM vol. 44 issue 6 Nov. 1997.

[Weinstock 97] Charles B. Weinstock and David P. Gluch. A Perspetive on the State of Research in Fault-Tolerant Systems. Special Report to Carnegie Mellon Software Engineering Institute.1997.

[Weisstein 99] Eric W. Weisstein et al. Extension Field. From MathWorld . http://mathworld.wolfram.com/ExtensionField.html.